A Brief Overview of C

If you've never had any exposure to C, this chapter contains most of what you'll need to know for this course. If you have had exposure to C, feel free to skim, but keep in mind that this chapter goes a bit deeper than most introductions to C.

Let's look at a very simple C program in *source code* form.

```
#include <stdio.h>

int main() {
  printf("Hello world!\n");
  return 0;
}
```

Type this program into an editor and save it with the name `helloworld.c`. I recommend typing the program instead of copying-and-pasting it because retyping it will force you to notice important details about the program.

Hopefully it's not too much of a stretch for you to figure out what this program does. We will look at this program line-by-line to understand what its parts are, but first, let's understand how to run this program.

The C Compiler

A computer cannot understand a C program in source code form. Source code is for humans to read and understand. In order for a computer to run a program in source code form, it must be translated into an equivalent, machine-readable form called an *executable binary*. An executable binary consists of *machine code* that looks a bit like this:

```
01111111 01000101 01001100 01000110 00000010 00000001
00000001 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000010 00000000
```

```
00111110 00000000 00000001 00000000 00000000 00000000
00110000 00000100 01000000 00000000 00000000 00000000
00000000 00000000 01000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00011000 00011010
...
```

Perhaps not surprisingly, we often call programs in machine-readable form "binaries" for short. The program that performs the translation from source code form to executable binary form is called a *compiler*. The C compiler translates C source code programs to machine code.

Note that there is a "human-readable" form of machine code called *assembly language* intended to make binary executables a little easier for humans to read, although reading them in this form is a difficult skill to attain. Each machine instruction is given a name, called an *instruction mnemonic*, and these mnemonics are printed instead of the binary. There is (generally) a one-to-one correspondence between assembly language mnemonics and machine instructions.

To give you a taste for what assembly looks like, here is `helloworld` compiled to x86 (Intel) assembly language. You do not need to understand assembly in this class!

```
.text
.file "helloworld.c"
.globl main
.align 16, 0x90
.type main,@function
main:                                   # @main
.cfi_startproc
# BB#0:
pushq %rbp
.Ltmp0:
.cfi_def_cfa_offset 16
.Ltmp1:
.cfi_offset %rbp, -16
movq %rsp, %rbp
.Ltmp2:
.cfi_def_cfa_register %rbp
subq $32, %rsp
movabsq $.L.str, %rax
movl $0, -4(%rbp)
movl %edi, -8(%rbp)
movq %rsi, -16(%rbp)
movq %rax, %rdi
movb $0, %al
callq printf
```

```
xorl %ecx, %ecx
movl %eax, -20(%rbp)         # 4-byte Spill
movl %ecx, %eax
addq $32, %rsp
popq %rbp
retq
.Lfunc_end0:
.size main, .Lfunc_end0-main
.cfi_endproc

.type .L.str,@object          # @.str
.section .rodata.str1.1,"aMS",@progbits,1
.L.str:
.asciz "Hello world!\n"
.size .L.str, 14


.ident "clang version 3.8.0-2ubuntu4 (tags/RELEASE_380/final)"
.section ".note.GNU-stack","",@progbits
```

History

Now that I've defined a few terms for you, let's briefly discuss some C history so that you can understand the importance of the language. Despite being more than 40 years old, C is still widely used.

C is a general-purpose programming language originally designed between 1969 and 1973 at Bell Labs by Dennis Ritchie. Its purpose was to make it easier to implement and maintain programs across a variety of computer architectures. In the early days of computing, *portability*, or the ability to easily move a program from one computer platform to another, was difficult. Often, each brand of computer had its own unique set of machine instructions and programming tools. *Porting* a program from one computer to another often meant that the entire program had to be rewritten. C was one of the first languages designed so that, as long as each target platform had a standard C compiler, a programmer needed to do to little more than run the C compiler in order to "port" their program.

(NOTE: Portability was more of a problem in the early days of computing when there were many different competing and incompatible computing platforms. Our modern computer ecosystem is dominated by two platforms, x86 and ARM. Furthermore, portable languages are

now the norm, so most programmers don't think much about this problem anymore.)

C is also an *imperative* language, meaning that in order to instruct a computer to do something, you need to tell it exactly what to do, step by step. As you will see this semester, many interesting programming languages are *not* imperative. C is also fairly "low-level,"[3]; meaning that a single instruction in a C program often closely corresponds with a single instruction in computer hardware. Consequently, C allows a degree of control over computer hardware that is not attainable by many other languages. Thus C is the language of choice for programs where low-level hardware control is essential, like operating systems. For example, Linux is written in C. In fact, C was explicitly designed with operating systems in mind. The first widely-used version of the UNIX operating system was written in C by Ritchie and his collaborator, Ken Thompson.

The success of C is partly because it gives programmers a simplified "model" of a computer, but not such a simple model that it is difficult to write high-performance code. In fact, as you will see, in C, memory is a resource that must be manually managed by the programmer. If you come from a Java or Python background, this idea will be foreign to you: neither language allows you to manually manage the computer's memory. Nonetheless, the rules for managing memory yourself are fairly simple, and with this feature you can write very fast code, control hardware directly, and interact with other low-level parts of your operating system that would not be possible otherwise.

[3] Ritchie considered C a "high-level" language, because by the standards of the time, it was! C looks nothing like machine code, and many old-timers considered C far too abstract to be able to produce fast code. Nowadays, writing low-level code is discouraged, because most high-performance language implementations are capable of producing more efficient code than assembly hand-written by even very good programmers. We will discuss this *level of abstraction* concept more during this semester.

A typographic convention for this course

Before we talk about compiling `helloworld.c`, take note of a convention that I will use throughout this course. When you see a line that looks like,

```
$ [whatever]
```

this indicates a command that you should run using the *command line interface* (CLI) on one of our lab machines. You can access the CLI by running the `Terminal` program on one of our lab machines. The `$` denotes the command-line prompt and should not be typed. Be aware that some lab machines use a different symbol than `$` for the command-line prompt, but the idea is the same.

Compiling using `clang`

For this class, we will be using the `clang` compiler. If you already know some C, you may be familiar with the alternative `gcc` compiler. We will be using `clang` instead because it supports more modern C features and it provides much better error messages than `gcc`.

To compile `helloworld.c`, type:

```
$ clang helloworld.c
```

If you made no mistakes when you typed in your program, `clang` will print nothing. This silent-on-success convention is a little counter-intuitive if you are new to UNIX, but you should remember that most UNIX programs work this way.

If `clang` prints an error message, go back and look carefully at your program to find your mistake and try again.

Once you have successfully compiled your program, you should see a file called `a.out` in your working directory. The following command lists the current directory and shows that I now have an `a.out` file.

```
$ ls -l
total 16
-rwxrwxr-x 1 dbarowy dbarowy 8664 Sep  2 13:08 a.out*
-rw-rw-r-- 1 dbarowy dbarowy   97 Sep  2 13:06 helloworld.c
```

Running the program

Note that all the C compiler does is convert the program into a binary executable. It does not actually *run* the program. To run the program, type

```
$ ./a.out
```

You should be rewarded with the text `Hello world!` printed on screen.

Don't speak gibberish

Imagine you're traveling to Greece. Since they speak Greek there, not English, you bring a little English-to-Greek phrase book with you. During your daily interactions with people, like asking where you might

find a good restaurant, where to rent a bicycle, what to do in the evenings, etc., you look up the phrase you want to use in your book, and you speak that phrase to a person. When they respond, you look up their response in the book and translate it back to English.

What you *do not do* is randomly choose phrases from the book and just say them. Why? Because doing so makes no sense. When you ask a Greek shopkeeper "Οι άχρωμες πράσινες ιδέες ύπνο θυμωμένα;" ("Do colorless green ideas sleep furiously?") they will, in all likelihood, politely shoo you out the door.

Writing a program is exactly like using a phrase book. The purpose of a program is to *communicate what you want* to a computer. Right now, you probably need to look up what you want to say using the C language documentation. Eventually, you will remember phrases and you won't have to look them up.

Do not copy and paste code snippets from the internet (e.g., Stack Overflow) without understanding them. For all you know, you are speaking gibberish to the computer, and in all likelihood, it will not do what you want. Stack Overflow is a wonderful resource—for learning how to solve problems. But to really solve a problem, you must understand the solution.

Let's *understand* the program we just typed in.

Library `include` statements

The first line,

```
#include <stdio.h>
```

tells the C compiler to use the `stdio` library.

What does this mean? Well, it turns out that C is actually quite a small and simple language. When people think about C programs they've written in the past, most of what they've done is use code that comes from C code libraries. Printing, as it turns out, is not a built-in feature of the C language! So in order to print things, we import the `stdio` library, which provides functionality for "standard input and output" (i.e., "standard I/O", often shortened to `stdio`).

We will talk about *how* the C compiler *links* imported library code to your program in a future lesson.

Function definitions

The next line,

```
int main() {
```

denotes the start of a *function definition*, and that definition continues until we reach the `}` character at the end of the program.

A function, or more precisely in C, a *program subroutine*, is a sequence of instructions that are packaged up into a unit. We package code in this manner so that we can reuse sequences of instructions without having to type them over and over again. Instead, we *call the function*, which has the same effect. Also, since we often want to run the same code with small variations, function definitions allow us to *parameterize* the function so that we can supply the varying values *when we call the function*.

This function, which is called `main`, has no parameters. It is important to know that the `main` function in your program is special. The reason is that when your computer attempts to run your compiled program, it needs to know where to begin running. That starting point, which is called an *entry point*, is always a function called `main`[4] in the C language.

Our `main` function also returns a value of type `int`. How do we know? The text to the left of the function name (in this case, `main`), denotes the *return type*. This means that the very last thing a function must do is return a value of the given type.

Finally, the "inside" of the function, what we call a *function body*, is the most important part. The function body is a sequence of instructions to perform. The key functionality of our `helloworld.c` program is located in the `main` function's body.

[4] This is actually a lie. The actual entry point is called `_start`, but the `_start` function is generated by the compiler and contains initialization code for the C language itself. From a programmer's standpoint, `main` really is the entry point.

Function calls

A *function call* tells the C compiler that you would like to use a function definition. If you define and never call a function, that function's body is never run.

A function call is performed by typing the name of the function followed by supplying values for its parameters in parenthesis. Suppose we have the following function definition:

```
int add(int x, int y) {
  return x + y;
}
```

We *call* the `add` function in our program with code that looks like:

```
add(3,4);
```

which will return the `int` value `7`.

"But wait," you protest, "we never call `main` in `helloworld.c`!"

Indeed, we never call `main`. As I noted before, `main` is a special function. When you run a program, the entry point is located and run, and in C, the entry point is the `main` function. Who calls `main`, then? The operating system calls `main` (or more precisely, the *program loader*).

Program statements

In C, a "line of code" must end in a semicolon. This construct is called a *program statement*. This is not unlike ending English sentences with periods– it tells you where the "end" of a sentence is, which helps with understanding. If you've even encountered a "run on sentence" in English, you know that sentences without periods are hard to understand. For the same reason, C statements must end in semicolons.

Note that other programming languages don't always use this semicolon convention. Instead, they have other ways to denote the end of a statement. Python, for example, is sensitive to indentation. We will see some other examples as the semester progresses.

Why don't some C constructs end in semicolons, like `#include` and function definitions? Because the C compiler knows when these constructs end without needing a semicolon. Admittedly, the rules seem a bit arbitrary to newcomers, but you'll eventually get the hang of them.

Printing

Now we get to the most important part of our program:

```
printf("Hello world!\n");
```

The `printf` function prints things to the screen. In this case, it prints "Hello world!" followed by a command, `\n`, that tells the computer to print a new line.

Recall that earlier, I stated that printing was not a feature of the C language, and here we are, printing. The reason we are able to print is because, earlier in the program, we told the C compiler to import the `stdio` library, which includes the `printf` function.

Note that this is an example of a function call. We supply the name of the function, `printf`, along with its parameter, in this case, the value `"Hello world!\n"`.

You might be wondering why the function is called `printf` instead of just `print`. The reason is that `printf` is short for "print formatted."

On-line help

This is a good time to mention that every UNIX-like computer, including the Linux and Mac machines we use in our labs come with a built-in help system called "manual pages," or "man pages" for short. Libraries like `stdio` are not a part of the C language. Technically they are a part of a separate collection of code called the "C Standard Library" and are supplied *with the operating system*. Practically speaking, no C compiler is shipped out to users without some kind of standard C library, because little can be achieved with such a language. Thus you can almost always

Section	Description
1	General commands
2	System calls
3	Library functions, particularly the C Standard Library
4	Special files
5	File formats
6	Games
7	Miscellaneous
8	System administration

Table 1: Sections of a `man` page.

count on the C standard library being available, with documentation, on a modern computer.

For example, on a lab machine, you can type the following into your CLI:

```
$ man 3 printf
```

and you will be rewarded with documentation for `printf`. What does the `3` mean? You need to tell `man` which "section" of the manual to search for `printf`. The sections are shown in table 1.

Since `printf` is a part of the C Standard Library, we type `man 3 printf` to find it. If you just type `man printf`, the help system will find a *different* `printf` command which is not the one you want.

Return value

Finally, we get to the penultimate line in the program,

```
return 0;
```

The `return` keyword instructs the function to return the following value. Since our function definition states that the return value of `main` is an `int`, the value we return must be an `int` or the compiler will print a compiler error.[5]

If you're like me, you might be wondering, "OK, I understand that we have to return an `int` because the `main` function definition says that we will. But *why* do we have to return an `int`? What does this `int` mean?"

Great question. The meaning of the return value of the `main` function is a signal to the operating system that the program either ran fine, or that it terminated with an error. Conventionally, the return value `0` means "returned without error." Any other number means that the program failed. Different operating systems have different meanings for non-zero return values.

The reason we use these return values for `main` is due to the design of the UNIX operating system: in UNIX we are encouraged to construct

[5] Note that compiler errors are a *feature* of a language, and even though they may seem annoying at times, they are very useful. Read them! They almost always correctly tell you what is wrong with your program. We will talk more about compiler errors—especially *type errors*—in more detail later in the semester.

complex programs out of less complex programs. If *another program* utilizes your `helloworld` program, it is important for that other program to know whether `helloworld` ran correctly or failed so that it can take the right action. We will not discuss the UNIX design much during this course, but if you are interested, I highly recommend taking a course in operating systems (or read "The Art of Unix Programming" by Eric S. Raymond, ISBN 0131429019). Understanding the design of UNIX will make you a better programmer.

One small detail: if you omit the return statement, specifically for the `main` function, the compiler will not complain, and will silently return `0`.

Compiler warnings

Earlier, we stated that you could compile a C program by typing

```
$ clang helloworld.c
```

and that, if the program contained no errors, `clang` would print nothing. It turns out that programs often have tiny flaws that are not crucial to the functioning of the program but which you really should consider fixing anyway. `clang` is capable of *warning* you when your code compiles but may not compile as you intend. To show warnings, add the `-Wall` flag:

```
$ clang -Wall helloworld.c
```

Now the compiler will print anything potentially problematic. `-Wall`, by the way, stands for "all warnings." For more information on warnings, type `man 1 clang` into your CLI.

In this class, your code must compile without warnings. If you turn in code that causes `clang` to print warnings when `-Wall` is used you will lose points on your homework grade!

Named compiler output

If `clang` is able to successfully compile your program, it will print nothing (in fact, it secretly returns `0` behind the scenes) and produce an executable binary called `a.out` on the side. With the `-o` option, `clang` lets you *name* this binary. For example,

```
$ clang -Wall -o helloworld helloworld.c
```

will run `clang` with warnings and will produce an executable binary called `helloworld` instead of `a.out`. This binary can be run with

```
$ ./helloworld
```

Makefiles

Typing commands like `clang -Wall -o helloworld helloworld.c` over and over again gets pretty tedious. And as your programs grow in complexity, you will need to type more complicated commands. There is a simple facility that is frequently (in fact, almost always) paired with a C language program: `make`. In this class, your C programs must always be accompanied by a *makefile*.

A makefile tells your C compiler how to build a program. Let's look at a simple example.

In your editor, create a file in the same directory as your `helloworld.c` program and call it `Makefile`. Type the following into the file:

```
helloworld: helloworld.c
↦clang -Wall -o helloworld helloworld.c
```

where ↦ represents a tab character.

Note that the space on the second line, before `clang`, *must be a real tab character, not a bunch of space characters*. If you are using `emacs` and you've named the file "`Makefile`", `emacs` will insert a real tab even if you've configured it to insert spaces instead of tabs. In other words, `emacs` does the right thing. Makefiles that do not have tabs will not run properly.

Now, on your command line, run

```
$ make helloworld
```

Assuming that your program has no errors, this will run `clang` and produce a new `helloworld` binary. Maybe.

Wait... "maybe"?

Make is a fairly smart utility. One of the things it does is to check whether you actually *need* to run `clang` again. If the `helloworld` binary is newer (i.e., has a more recently modification date) than `helloworld.c`, then by default, make will not bother running the command again.

Since computers are relatively fast, you might be wondering why `make` bothers to do this. For our short `helloworld.c` program, the time saved makes almost no difference. The real benefit of `make` starts to become apparent when we add multiple *rules*.

`make` rule

As it stands, our `Makefile` currently only has a single rule, called `helloworld`. A rule is composed of a *target*, a *dependency list*, and a *command list*. Rules

have the following syntax:

```
<target name>: <dependency 1> ... <dependency m>
↦<command 1>
↦...
↦<command n>
```

The *target* is the name of the rule. Generally speaking, your target name should be the same as the name of the file that you want to produce. In our `helloworld` target, we have a single `clang` command that builds a `helloworld` binary.

The target name is how `make` knows to look at the modification date for the `helloworld` file. But how does it know what to compare `helloworld` against? This is where dependencies come in.

`make` dependencies

Dependencies tell `make` which file or files your target depends on. In our case, we want to update the `helloworld` binary when the `helloworld.c` source file changes. `helloworld.c` is our sole dependency. You can list as many dependencies as you want, separated by spaces.

You may specify other make targets as dependencies. To demonstrate, let's change how we compile `helloworld.c`. Instead of converting the C program to an executable binary all at once, let's instead convert the C program to assembly language, and then convert the assembly language to a binary in a separate step. To be clear, compiling `helloworld.c` in two steps is not strictly necessary; I am showing this as two steps just to make it clear how `make` dependencies work.

Rewrite your `Makefile` as:

```
helloworld: helloworld.s
↦clang -o helloworld helloworld.s

helloworld.s: helloworld.c
↦clang -Wall -S helloworld.c
```

Now, if you type `make helloworld`, `make` will produce an assembly language file called `helloworld.s` before producing the `helloworld` binary. If you look at the `helloworld.s` file in a text editor, you should see something that looks very much like the assembly program shown earlier in this document.

How does `make` know that it should produce a `helloworld.s` before producing a `helloworld` file? Because you told it so: the dependency for `helloworld` is `helloworld.s`.

The `make` algorithm

When you run `make helloworld`, `make` checks that `helloworld.s` exists and is older than `helloworld`. If not, it moves on to the `helloworld.s` target, otherwise, it stops.

`make` now checks that `helloworld.c` exists and is older than `helloworld.s`. If not, it looks for a rule called `helloworld.c`. Since the file `helloworld.c` always exists, `make` will only run the command in the `helloworld.s` target when `helloworld.c` is newer than `helloworld.s`. After running the command, the `helloworld.s` file exists.

Now `make` returns to the `helloworld` target, finally producing the `helloworld` binary.

`make` dependencies are a DAG

An astute student (especially if you've taken CSCI 136!) should recognize that the chain of dependencies in a makefile can be represented as a graph. Each make target is a vertex in a graph, and each dependency is an edge from the target vertex to the dependency, which is also a vertex. In fact, this graph *must* be a directed acyclic graph (DAG), otherwise `make` will not work properly.

Figure 3 shows the DAG for our `helloworld` makefile thus far.

Figure 3: A directed acyclic graph representing helloworld dependencies.

Thinking about a `makefile` as a graph is very useful for understanding what `make` will do. If you are confused about a `makefile`, I strongly recommend drawing the graph out on paper.

Default `make` target

With our current `Makefile`, we don't actually have to type `make helloworld`. In fact, we can just type

```
$ make
```

and it will also work. Why?

If you call make without a target name, it will run the *first* target in the file. The first target is called the *default target*. The default target should generally be the file that you want to produce most often, i.e., the executable binary.

In fact, you can call *any* make target on the command line. If you type:

```
$ make helloworld.s
```

Then you are asking `make` to produce *only* the `helloworld.s` file (and any other dependencies that may need to be produced to create `helloworld.s`).

"Cleaning"

It is often useful to "clean up" the files created during development so that only the essential files remain. In our case, the only essential file is `helloworld.c`. We can generate `helloworld.s` and `helloworld` anytime we want by running `make`. In ordinary software development that uses a build system like `make`, it is considered polite to always provide a `clean` target. In general, `clean` should remove all temporary files produced during compilation.

If you use `emacs`, you probably also produce many files like `helloworld.s` as a side-effect. These files are temporary save files produced by `emacs` in case your computer crashes while you are working on a file. They allow you to restore your work in case you forgot to save. This is definitely useful, but I also like to delete these files when I clean up, because they add a lot of clutter to my source code folder.

Let's add a `clean` target to our makefile. Put the following at the bottom:

```
.PHONY: clean
clean:
↦rm -f helloworld helloworld.s *~
```

When we run `make clean`, `make` will delete those files. We supply the `-f` flag with `rm` in case files don't exist. If, for example, `helloworld` exists but `helloworld.s` does not, technically `rm` will notice that `helloworld.s` is missing and terminate with an error. `-f`, which means "force deletion", tells `rm` to ignore those missing files.

One last thing: notice that our `clean` target does not have any dependencies. When `make` encounters a no-dependency target, it will simply run the commands listed in the rule without doing any dependency modification-time checks. However, the convention in `make` is that the target refers to a filename. What if you just so happen to have a file in your directory called `clean`? The short answer is that `make` will refuse to clean, because it sees that a file called `clean` already exists. To let `make` know that it shouldn't bother checking, that `clean` is a kind of "phony file," we write `.PHONY: clean`. Then if a `clean` file exists, the `clean` target can still be reliably run.

`all` rule

Sometimes a makefile is a collection of rules for separate programs (e.g., a homework assignment consisting of solutions to multiple problems). It is often convenient to create a single rule that builds all of the targets. Conventially, this rule is called `all` and has only dependencies, no commands. For example,

```
all: problem1 problem2 problem3

problem1: problem1.c
↦clang -Wall -o problem1 problem1.c

problem2: problem2.c
↦clang -Wall -o problem2 problem2.c

problem3: problem3.c
↦clang -Wall -o problem3 problem3.c
```

Notice that in the sample makefile above, `all` is the first rule, so

```
$ make all
```

and

```
$ make
```

do the same thing.

More C

Let's explore some more features of the C language. Since you likely have been exposed to Java before, C will look visually similar to you. In fact, Java was explicitly designed to resemble C to encourage C programmers to try it out. This was a very successful tactic, and it is one of the reasons why Java is more popular than C now.

Keep in mind, however, that C is not Java. In fact, Java is much more sophisticated than C, and Java does a lot more work behind the scenes to ensure that your program does what you *want*. C lacks many of these safeguards.

Comments

In C, there are two kinds of comments: single-line comments and multi-line comments. They work exactly the same way as their Java equivalents.

```
// This is a single-line comment.

/* This is a
   multi-line comment. */
```

Primitive	Description
`char`	The smallest addressable unit of the machine that can contain an element of the ASCII character set.
`int`	A signed integer.
`float`	An IEEE 754 single-precision binary floating point number.
`double`	An IEEE 754 double-precision binary floating point number.

Table 2: C primitive data types.

Operator	Description	Example	Evaluates To
`+`	Addition	`2 + 2`	`4`
`-`	Subtraction	`2 - 2`	`0`
`*`	Multiplication	`2 * 2`	`4`
`/`	Division	`2 / 2`	`1`
`%`	Modulus	`2 % 2`	`0`

Table 3: C infix operators.

Variables

As with Java, C has variables. The statement

```
int i = 0;
```

does essentially the same thing in Java as it does in C. First, storage for the variable `i`, which is of type `int`, is *allocated*. Then the integer value `0` is *assigned* to that location. We will talk about allocation and assignment in much more detail when we talk about how C deals with computer memory. For now, remember that using a variable properly always consists of two steps:

1. *Allocation* is the mechanism by which a C program obtains memory.
2. *Assignment* is the mechanism by which a C program stores a value in a memory location.

In C, you must always think about *where* a variable is allocated.[6] In the code snippet above, `i` is what we call an *automatic variable*, because we did not explictly say anything about the *storage duration* for `i`. For now, keep in mind that, if you don't explicitly ask C to change the kind of storage, a variable's storage duration is "automatic."

I am intentionally leaving some of the terms here undefined because memory management in C is a complex topic. We will discuss these terms in detail when we cover memory management in C.

[6] I would argue that this is the most important fact to know about C and what causes the vast majority of C bugs. Watch out!

Arithmetic expressions

Like Java, C has a variety of infix arithmetic operators, as shown in table 3.

The rules for these operators are much like the rules in Java. For example, `3 / 4` equals `0` but `3 / 4.0` equals `0.75`. If you don't remember

Operator	Description	Example	Evaluates To
`+`	Unary plus	`+2`	`2`
`-`	Negation	`-2`	`-2`
`++`	Preincrement	`i = 0; ++i;`	Returns `1`, sets `i` to `1`
`--`	Predecrement	`i = 0; --i;`	Returns `-1`, sets `i` to `-1`
`++`	Postincrement	`i = 0; i++;`	Returns `0`, sets `i` to `1`
`--`	Postdecrement	`i = 0; i--;`	Returns `0`, sets `i` to `-1`

Table 4: C unary operators.

why, this would be a good time to brush up on your knowledge of integer and floating point data types.

C also has unary operators, as shown in table 4.

Primitive data types

C has a small set of data types that are referred to as *primitive*. Primitive data types are data types that are defined by the language–you cannot modify them. Furthermore, in C, primitive data types often correspond closely with the facilities afforded by specific hardware instructions. The primitives available in C are shown in Table 2.

Many of these primitives may also be *modified* using keywords like `signed` or `short` to specify different number ranges or sizes.

Quite surprisingly, C traditionally does not have a built-in `boolean` data type! In C, the `int` value `0` is used to represent `false` and *any other integer value* represents `true`. This is often confusing to people who come to C from more featureful languages, so for this class, I will allow you to use a modern version of C. In C99 and later, the C Standard Library has a boolean data type that you can use. You will need to `#include <stdbool.h>` to use it.

```
#include <stdbool.h>

int main() {
  bool b = true;
}
```

`clang` uses `C11` by default, so `stdbool` is available by default (yes, confusingly C11 is newer than C99).

Note that there is no mention here about other types you often see in Java like `String` and other *complex* data types like classes. C has no strings and no classes. It does however, have two facilities for building complex data types.

Structures

Complex data types (i.e., data types that allow a variable to store more than one primitive value) in C are achieved using a feature called a *structure*, or a `struct` for short.

A `struct` vaguely resembles a class in Java. For example,

```
struct point {
    int x;
    int y;
};
```

The above `struct` definition defines a new type called `point` that stores two integers, one called `x` and another called `y`. Note that C requires you to put a semicolon (`;`) after the `struct` definition.[7]

[7] I always forget to do this!

To use our `point`, we first need to allocate storage in a variable:

```
struct point p;
```

Again, since we did not say anything "special" about the storage, `p` is an automatic variable.

To assign values to `p`, we use the *field access operator*, `.`, as follows:

```
p.x = 3;
p.y = 4;
```

Note that, unlike Java classes, a `struct` does not have methods or a constructor. It also does not have field access modifiers such as `public`, `private`, and so on. It is simply a container for data.

Arrays

Arrays in C are similar to Java arrays in that they are a fixed-size data structure that stores a sequence of elements, and they allow random-access reads and writes.

Here's some code for allocating an array, assigning values to it, and then reading and printing them back out.

```
/* Allocate, assign, read an array in C */
int arr[10];

for(int i = 0; i < 10; i++) {
  arr[i] = i * 2; // store the value of i * 2 in the array at index i
}

for(int i = 0; i < 10; i++) {
  printf("%d\n", arr[i]); // print the values out
}
```

Observe that the syntax for allocating an array in C is also a little different than in Java.

Unfortunately, because C is not object-oriented like Java, working with arrays is a tad trickier in some cases. Remember that C does not have classes, so types do not have members. In Java, you can "ask" an array how long it is by doing

```
/* Allocate array and get length in Java */
int[] arr = new int[10];
int len = arr.length;
System.out.println(len);
```

`length` here is a member function on the Java array data type. In C, it is not simple to perform this operation because there are no member functions. Instead, you need to either 1) remember the length you used when you allocated the array, or 2) use the C `sizeof` operator.

Let's look at the `sizeof` operator. The `sizeof` operator gives the amount of storage, in bytes, required to store a value for a variable of a given type. So the output of `sizeof` for an `int` array of size `10` is, surprisingly:

```
/* Using sizeof in C */
int arr[10];
printf("\%lu\n", sizeof(arr)); // prints '40'
```

Why? Because an `int` is 4 bytes (on my machine). Storing 10 `int`s, one after the other, takes up 10*4 bytes = 40 bytes.

This means that if we want to find out the number of elements in an array, we need to do a little work:

```
/* Allocate array and get length in C */
int arr[10];
int len = sizeof(arr) / sizeof(int); // 40 / 4 = 10
printf("%lu\n", len);
```

Of course, we could have just saved the value `10` from when we allocated the array.

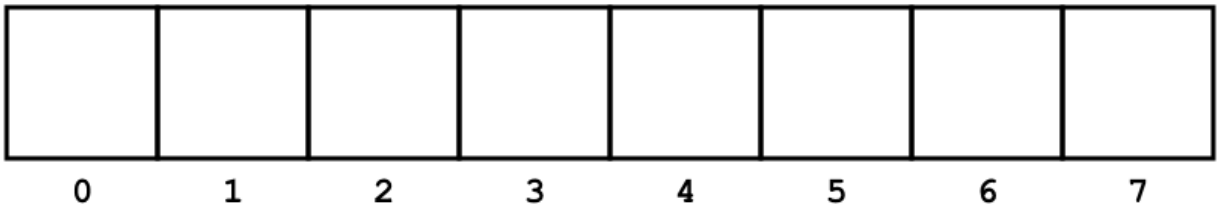
Strings

C does not have a string data type. You might be wondering, then, how on earth people write programs in C that have anything to do with text.

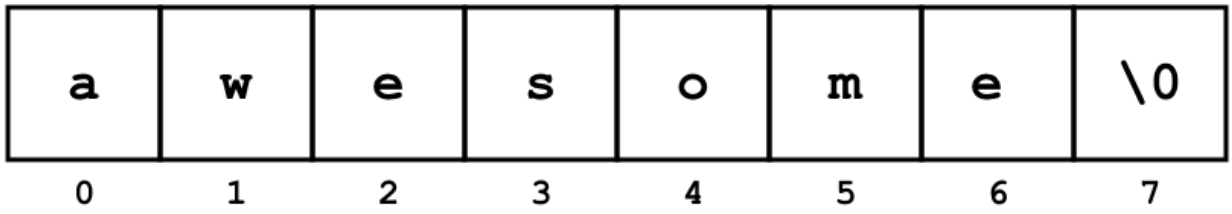
In C, we use arrays to represent strings. In most other languages, strings are indeed represented using array "under the hood," so this isn't dramatically different from the computer's standpoint. Be aware that the language is completely unaware of strings– from the compiler's perspective, they're just arrays. Conventionally, however, what has become known as the "C string" convention requires you to follow two rules:

1. A C string is an array of characters.
2. Every C string must be NULL-terminated.

What does this mean? Think of an array:



The C string "awesome" is represented as



Notice that the array must be big enough to store the NULL character, `0`, at the end. *Without a terminating null character, a chararacter array is NOT a C string!*

The C Standard library comes with a set of functions that make working with C strings a little less cumbersome. Be aware that if your strings are not NULL-terminated, most of these functions will misbehave.[8] You can use the C string functions with

[8] In fact, C string bugs are a major source of security vulnerabilities in software written in C. You should *never* use the `strcpy`, `strcat`, and `gets` functions. Most modern C compilers will warn you to consider an alternative if you do.

```
#include <string.h>
```

Remember that anything you do with strings in C must use these functions. For example, the following expressions will probably not do what you want:

```
char s1[8] = "awesome";
char s2[8] = "awesome";

bool b = s1 == s2;    // always false
s2 = "not awesome?";  // cannot assign to s2; does not compile
s1 = s1 + "ish";      // + not defined on arrays; does not compile
```

Let's look at a simple program that reads in your name and birthday, if your birthday is today, tells you "happy birthday!".

```
#include <stdio.h>
#include <string.h>
#include <time.h>

int main() {
  char fname[100];
  char month[20];
  char day[20];
  char month_today[20];
  char day_today[20];

  // today's date
  time_t t = time(NULL);
  struct tm *tm = localtime(&t);

  // convert today's date to C strings
  strftime(month_today, 20, "%B", tm);
  strftime(day_today, 20, "%-e", tm);

  // read name
  printf("What is your first name? ");
  fgets(fname, sizeof(fname), stdin);
  fname[strcspn(fname, "\n")] = '\0';

  // read birth month
  printf("What month were you born? ");
  fgets(month, sizeof(month), stdin);
  month[strcspn(month, "\n")] = '\0';

  // read birth day
```

```
  printf("What day were you born? (1-31) ");
  fgets(day, sizeof(day), stdin);
  day[strcspn(day, "\n")] = '\0';

  // compare dates
  if (strncmp(month, month_today, 20) == 0 &&
      strncmp(day, day_today, 20) == 0) {
    printf("Happy birthday, %s!\n", fname);
  }
}
```

There's a lot you probably have not seen here before. That's OK! We'll go through the important parts now.

At the beginning of the program, we allocate storage for a number of C strings: the user's first name, month and day of birth, and today's month and day.

We then compute today's date using `time` and `localtime`, and we convert the output of `localtime` to C strings using `strftime`. We are not going to talk about these just yet, since they involve pointers, but if you're curious, look them up using `man 3 time`, etc.

After prompting the user for their name, we read what they type in using the `fgets` call. `fgets` takes the destination array ("buffer" in C-speak) as the first parameter, the maximum number of bytes to read (so we use `sizeof`), and where we want to read from (in this case, standard input or `stdin`). You'll notice the odd-looking line

```
fname[strcspn(fname, "\n")] = `\0`;
```

right after. What problem do you think this line solves? Try running the above program with and without that line. What happens? How does `strcspn` solve the problem?

Finally, we compare the dates. Since C knows nothing about C strings, we cannot use a simple == to compare them. Instead, we use the `strncmp` function. `strncmp` takes two arrays and the maximum number of characters to compare.

This program still leaves a lot to be desired. For example, it happily accepts the following inputs:

```
What is your first name? Daniel
What month were you born? Octember
What day were you born? (1-31) 67
```

You can find documentation for all the C string functions by typing `man 3 string`.

String Literals

Literal values are fixed values supplied with the source code of a program. For example.

```
double pi = 3.14159265359;
```

C has special support for string literals, since they are used often, just as they are in Java. The following is also a literal.

```
char *msg = "Hello, everyone!";
```

(we will discuss the meaning of the type `char *` soon)

You can use string literals in much the same way that you use character arrays in C (in fact, they *are* character arrays), with one critical exception: string literals are *read only*. That means, if you take the following program:

```
char *msg = "You all everybody!\n";
printf("\%s", msg);
```

and modify it (all we're doing here is copying the string from its current location back to its current location)

```
char *msg = "You all everybody!\n";
strcpy(msg, msg, strlen(msg));
printf("\%s", msg);
```

trying to run it will result in

```
Segmentation fault (core dumped)
```

Since string literals are usually stored in read only memory, you are not allowed to update them. A *segmentation fault* is an error that occurs when your program attempts to access memory with an operation that is not allowed.

Printing, again

Let's dig into the `printf` statement in a little more detail. As stated before, `printf` is for printing.

`printf` takes at least one argument, but may take many more. The first argument is called the *format string*. The format string *must* be a string literal. For example,

```
printf("Hello world!\n");
```

Format Specifier	Purpose
`%c`	a single character
`%d`	an `int`, printed as a decimal (base 10) number
`%u`	an `unsigned int` (aka `uint`) printed as a decimal number
`%f`	a floating point number
`%s`	a C string
`%x`	an `int`, printed as a hexadecimal (base 16) number
`%o`	an `int`, printed as an octal (base 8) number
`%`	a literal percent sign

Table 5: Some C format specifiers.

But `printf` is more powerful than this. `printf` can also perform *string interpolation*, which will substitute other text in for placeholders you put into the format string. The manner in which this substitution is performed depends on the kind of placeholder you use. This is why placeholders are called *format specifiers*.

For example.

```
char *name = "Dan";
printf("Hello %s!\n", name);
```

Here we're asking `printf` to substitute the variable `name` where the `%s` format specifier appears. You can put as many format specifiers in the format string as you like, as long as you supply enough values to `printf` to do the substitution.

```
char *name = "Dan";
char *town = "Williamstown";
char *state = "Amazing Commonwealth of Massachusetts";
printf("Hello %s, who lives in %s in the %s", name, town, state);
```

Choosing the appropriate format specifier depends on the 1) type of the data you want to print, and 2) the manner in which you want it printed. Above, we used `%s`, which is for printing C strings. A summary of the most useful format specifiers is shown in Table 5.

You can also do a variety of useful formatting transformations, like printing with a lower precision. For example,

```
double pi = 3.14159265359;
printf("\%.4f\n", pi);
```

prints `3.1416` to the screen. Note that the last digit is rounded up. Rounding rules for floating point numbers follow the rules specified by the IEEE 754 floating point standard.

See `man 3 printf` for more information.

Control constructs

C has the same control constructs that Java has: `for` and `while` loops, and `if` and `else` conditionals.

A `for` loop:

```
printf("I'm not listening to you ");
for(int i = 0; i < 1000; i++) {
  printf("LA");
}
printf("\n");
```

A `while` loop:

```
char c = 'n';
while(c != 'y') {
  printf("Are you annoyed yet? y/n ");
  c = getchar();
  fpurge(stdin);
}
```

(Think about why I am able to compare `c` with `'y'` even though I said that C does not support comparison of strings. Also, what does `fpurge` do?)

A conditional:

```
if (1 == 2) {
  printf("Bad things are happening.");
} else {
  printf("Well OK, then.");
}
```

Anything else?

I know what you're thinking. "Please promise me that we're done talking about C." Fortunately, C really is a simple language, and the above syntax is almost all you need to know. However, most C programs rely heavily on pointers, and for that reason, we'll spend more time talking about using pointers effectively. Don't be frightened! Pointers have a reputation for being scary,[9] but the reputation is undeserved. They are actually quite simple, and you'll see in our next reading.

[9] Pointers themselves are not scary. What's scary are the bugs that an undisciplined use of pointers can cause.